Programación Concurrente

Clase 7

Facultad de Informática

UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Pasaje de Mensajes Sincrónicos (PMS): https://drive.google.com/uc?id=1xJS1-HKs6FMWRfH6tprV2vZ0Nu-2Hj0R&export=download

- CSP – Lenguaje para PMS: https://drive.google.com/u/1/uc?id=1TgzNTMds7QPzHpLaYY4LjzM9lOVeB9Xx&export=download

- Paradigmas de Interacción entre Procesos: https://drive.google.com/uc?id=1a0QUfXjpdM26pTIGzSEjm7zGRtCxAs9d&export=download

Pasaje de Mensajes Sincrónicos (PMS)

Diferencia con PMA

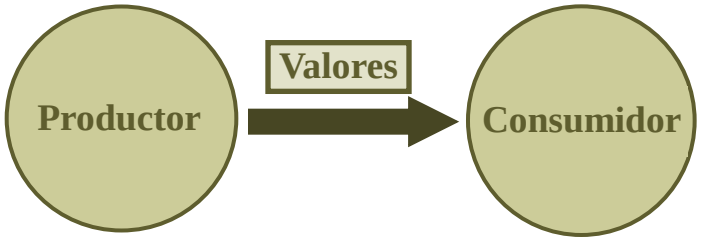
- Los canales son de tipo ***link*** o punto a punto (1 emisor y 1 receptor).
- La diferencia entre *PMA* y *PMS* es la primitiva de trasmisión ***Send***. En *PMS* es bloqueante y la llamaremos (por ahora) ***sync_send***.
  - El trasmisor queda esperando que el mensaje sea recibido por el receptor.
  - La cola de mensajes asociada con un *send* sobre un canal se reduce a 1 mensaje $\Rightarrow$ **menos** memoria.
  - Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA *(los emisores se bloquean)*.
- Si bien ***send*** y ***sync_send*** son similares (en algunos casos intercambiables) la semántica es diferente y las posibilidades de ***deadlock*** son mayores en comunicación sincrónica.

Ejemplo: *productor / consumidor*

```
chan valores(int);

Process Productor
{  int datos[n];
   for [i=0 to n-1]
      {  #Hacer cálculos productor
         sync_send valores (datos[i]);
      }
}

Process Consumidor
{  int resultados[n];
   for [i=0 to n-1]
      {  receive valores (resultados[i]);
         #Hacer cálculos consumidor
      }
}
```



Comentarios de PMS

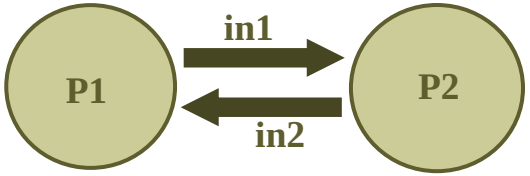
- Si los cálculos del productor se realizan mucho más rápido que los del consumidor en las primeras ***n/2*** operaciones, y luego se realizan mucho más lento durante otras ***n/2*** interacciones:
  - Con PMS los pares send/receive se completarán asumiendo la demora del proceso que más tiempo consuma. Si la relación de tiempo fuera 10 a 1 significaría multiplicar por 10 los tiempos totales.
  - Con PMA, al principio el productor es más rápido y sus mensajes se encolan. Luego el consumidor es más rápido y "descuenta" tiempo consumiendo la cola de mensajes.
- ***Mayor concurrencia en PMA***. Para lograr el mismo efecto en PMS se debe interponer un proceso ***"buffer"***.
- ¿Que pasa si existe más de un productor/consumidor?.

Comentarios de PMS

- La concurrencia también se reduce en algunas interacciones Cliente/Servidor.
  - Cuando un cliente está liberando un recurso, no habría motivos para demorarlo hasta que el servidor reciba el mensaje, pero con PMS se tiene que demorar.
  - Otro ejemplo se da cuando un cliente quiere escribir en un display gráfico, un archivo u otro dispositivo manejado por un proceso servidor. Normalmente el cliente quiere seguir inmediatamente después de un pedido de write.
- Otra desventaja del PMS es la mayor probabilidad de ***deadlock***. El programador debe ser cuidadoso de que todas las sentencias ***send*** (sync_send) y ***receive*** hagan matching.

Deadlock en PMS

- Dos procesos que intercambian valores.




```
chan in1(int), in2(int);

Process P1
{    int valorA = 1, valorB;
     sync_send in2(valorA);
     receive in1(valorB);
}

Process P2
{    int valorA, valorB = 2;
     sync_send in1(valorB);
     receive in2 (valorA);
}
```

```
chan in1(int), in2(int);

Process P1
{    int valorA = 1, valorB;
     sync_send in2(valorA);
     receive in1(valorB);
}

Process P2
{    int valorA, valorB = 2;
     receive in2 (valorA);
     sync_send in1(valorB);
}
```

CSP– Lenguaje para PMS

El lenguaje CSP (Hoare, 1978)

- CSP (Communicating Sequential Processes, Hoare 1978) fue uno de los desarrollos fundamentales en Programación Concurrente. Muchos lenguajes reales (OCCAM, ADA, SR) se basan en CSP.
- Las ideas básicas introducidas por Hoare fueron PMS y ***comunicación guardada***: PM con waiting selectivo.
- ***Canal***: link directo entre dos procesos en lugar de mailbox global. Son *half-duplex* y nominados.
- Las sentencias de Entrada (**?** o **query**) y Salida (**!** o **shriek** o **bang**) son el único medio por el cual los procesos se comunican.

  process A { . . . B ! e; . . . } **process B { . . . A ? x; . . . }**

- Para que se produzca la comunicación, deben ***matchear***, y luego ***se ejecutan simultáneamente***.
- Efecto: sentencia de ***asignación distribuida***.

El lenguaje CSP (Hoare, 1978)

- Formas generales de las sentencias de comunicación:

Destino ! port(e_1, ..., e_n);

Fuente ? port(x_1, ..., x_n);

  - *Destino* y *Fuente* nombran un proceso simple, o un elemento de un arreglo de procesos. *Fuente* puede nombrar *cualquier* elemento de un arreglo (*Fuente*[*]).
  - ***port*** son etiquetas que se usan para distinguir entre distintas clases de mensajes que un proceso podría recibir (puede omitirse si es sólo uno).
  - Dos procesos se comunican cuando ejecutan sentencias de comunicación que hacen ***matching***.

***A* ! canaluno(dato);** ** *B* ? canaluno(resultado);**

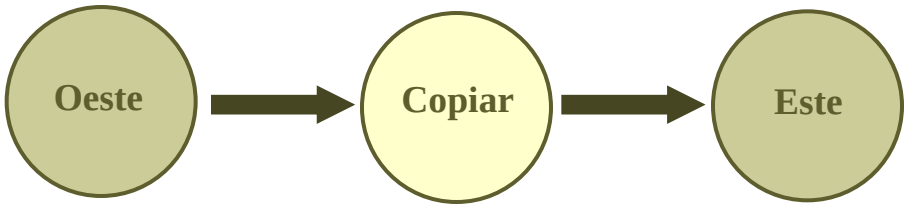
Ejemplos básicos

- *Proceso* filtro que copia caracteres recibidos del proceso *Oeste* al proceso *Este*:

```
Process Oeste
{   char c;
    do true →
            Generar(c);
            Copiar ! (c);
    od
}
```

```
Process Copiar
{   char c;
    do true →
            Oeste ? (c) ;
            Este ! (c);
    od
}
```

```
Process Este
{   char c;
    do true →
            Copiar ? (c) ;
            Usar(c);
    od
}
```



Ejemplos básicos

- Server que calcula el MCD de dos enteros con el algoritmo de Euclides. ***MCD*** espera recibir entrada en su port ***args*** desde un cliente. Envía la respuesta al port ***resultado*** del cliente.

```
Process MCD
{   int id, x, y;
    do true →
            Cliente[*] ? args(id, x, y);
            do  x > y → x := x - y;
            □   x < y → y := y - x;
            od
            Cliente[id] ! resultado(x);
    od
}
```

- *Cliente[i]* se comunica con *MCD* ejecutando:

```
...
MCD ! args(i, v1, v2);
MCD ? resultado(r);
...
```

Comunicación Guardada

- Limitaciones de **?** y **!** ya que son bloqueantes. Hay problema si un proceso quiere comunicarse con otros (quizás por ≠ ports) sin conocer el orden en que los otros quieren hacerlo con él.
- Por ejemplo, el proceso Copiar podría extenderse para hacer buffering de k caracteres: si hay más de 1 pero menos de k caracteres en el buffer, Copiar podría recibir otro carácter o sacar 1.

- Las operaciones de comunicación (**?** y **!**) pueden ser ***guardadas***, es decir hacer un AWAIT hasta que una condición sea verdadera.
- El **do** e **if** de CSP usan los *comandos guardados* de Dijkstra (B → S).

Comunicación Guardada

Las sentencias de comunicación guardada soportan comunicación no determinística:

$$B; C \rightarrow S;$$

- ***B*** puede omitirse y se asume *true*.
- ***B*** y ***C*** forman la guarda.
- La guarda tiene éxito si ***B*** es *true* y ejecutar ***C*** no causa demora.
- La guarda falla si ***B*** es *falsa*.
- La guarda se bloquea si ***B*** es *true* pero ***C*** no puede ejecutarse inmediatamente.

Comunicación Guardada

Las sentencias de comunicación guardadas aparecen en ***if*** y ***do***.

if B_1***; comunicación***$_1 \rightarrow S_1$***;***
- B_2***; comunicación***$_2 \rightarrow S_2$***;***

fi

Ejecución:

- *Primero*, se evalúan las guardas.
  - Si todas las guardas fallan, el if termina sin efecto.
  - Si al menos una guarda tiene éxito, se elige una de ellas (no determinísticamente).
  - Si algunas guardas se bloquean, se espera hasta que alguna de ellas tenga éxito.
- *Segundo*, luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación de la guarda elegida.
- *Tercero*, se ejecuta la sentencia S_i.

La ejecución del do es similar (se repite hasta que todas las guardas fallen).

Comunicación Guardada

Podemos re-programar Copiar para usar comunicación guardada:

```
Process Copiar
    {   char c;
        do Oeste ? (c) → Este!(c);
        od
    }
```

Extendemos *Copiar* para manejar un buffer de tamaño 2. Luego de ingresar un carácter, el proceso que copia puede estar recibiendo un segundo carácter de Oeste o enviando uno a Este.

```
Process Copiar2
    {   char c1, c2;
        Oeste ? (c1);
        do Oeste ? (c2) → Este ! (c1) ;
                          c1=c2;
        □  Este ! (c1) → Oeste? (c1);
        od
    }
```

Ejemplo

Copiar con un buffer limitado

```
Process Copiar
{  char buffer[80];
   int front = 0, rear = 0, cantidad = 0;
   do cantidad <80; Oeste?(buffer[rear]) → cantidad = cantidad + 1;
                                           rear = (rear + 1) MOD 80;
   □  cantidad >0; Este!(buffer[front]) → cantidad := cantidad - 1;
                                          front := (front + 1) MOD 80;
   od
}
```

- Con ***PMA***, procesos como *Oeste* e *Este* ejecutan a su propia velocidad pues hay buffering implícito.
- Con ***PMS***, es necesario programar un proceso adicional para implementar buffering si es necesario.

Ejemplo

Asignación de Recursos

```
Process Alocador
{   int disponible = MaxUnidades;
    set unidades = valores iniciales;
    int indice, idUnidad;

    do disponible >0;  cliente[*] ? acquire(indice) → disponible = disponible -1;
                                                      remove (unidades, idUnidad);
                                                      cliente[indice] ! reply(idUnidad);
    □  cliente[*] ? release(indice, idUnidad) →  disponible = disponible +1;
                                                 insert (unidades, idUnidad);
    od
}
```

La solución es concisa. Usa múltiples ***ports*** y un brazo del ***do*** para atender cada una. Se demora en un mensaje ***acquire*** hasta que haya unidades, y no es necesario salvar los pedidos pendientes.

Ejemplo

Intercambio de Valores

```
Process P1
{    int valor1 = 1, valor2;
     if  P2 ! (valor1) → P2 ? (valor2);
     □ P2 ? (valor2) → P2 ! (valor1);
     fi
}

Process P2
{    int valor1 , valor2 = 2;
     if  P1 ! (valor2) → P1 ? (valor1);
     □ P1 ? (valor1) → P1 ! (valor2);
     fi
}
```

Esta solución simétrica **NO** tiene *deadlock* porque el no determinismo en ambos procesos hace que se acoplen las comunicaciones correctamente. Si bien es simétrica, es más compleja que la de PMA...

Ejemplo

*La Criba de Eratóstenes para la g*eneración de números primos

- **Problema:** generar todos los primos entre 2 y $n \rightarrow 2\ 3\ 4\ 5\ 6\ 7\ 8\ \ldots\ N$
- Comenzando con el primer número (2), recorremos la lista y borramos los múltiplos de ese número. Si *n* es impar: $2\ 3\ 5\ 7\ \ldots\ n$
- Pasamos al próximo número (3) y borramos sus múltiplos.
- Siguiendo hasta que todo número fue considerado, los que quedan son todos los primos entre 2 y *n*.
- La criba *captura* primos y *deja caer* múltiplos de los primos.
- ***¿Cómo paralelizar?*** Pipe de procesos filtro: cada uno recibe un stream de números de su predecesor y envía un stream a su sucesor. El primer número que recibe es el próximo primo, y pasa los *no múltiplos*.

Ejemplo

*La Criba de Eratóstenes para la g*eneración de números primos

```
Process Criba[1]
{    int p = 2;

     for [i = 3 to n by 2]  Criba[2] ! (i);
}

Process Criba[i = 2 to L]
{    int p, proximo;

     Criba[i-1] ? (p);
     do Criba[i-1] ? (proximo) →
                  if  ((proximo MOD p) <> 0 ) and (i < L) →  Criba[i+1] ! (proximo);
     od
}
```

- El número total de procesos *Criba* (*L*) debe ser lo suficientemente grande para garantizar que se generan todos los primos hasta *n*.
- Excepto *Criba*[1], los procesos terminan bloqueados esperando un mensaje de su predecesor. Cuando el programa para, los valores de *p* en los procesos son los primos. Puede modificarse con centinelas.

Ejemplo

Ordenación de un Arreglo

- **Problema:** ordenar un arreglo de n valores en paralelo (n par, orden no decreciente).
- Dos procesos *P1* y *P2*, cada uno inicialmente con ***n/2*** valores (arreglos ***a1*** y ***a2*** respectivamente).
- Los ***n/2*** valores de cada proceso se encuentran ordenados inicialmente.
- **Idea:** realizar una serie de intercambios. En cada uno *P1* y *P2* intercambian ***a1[mayor]*** y ***a2[menor]***, hasta que *a1[mayor] < a2[menor]*.

Ejemplo

Ordenación de un Arreglo

```
Process P1
{  int nuevo, a1[1:n/2]; const mayor = n/2;
   ordenar a1 en orden no decreciente
   P2 ! (a1[mayor]);
   P2 ? (nuevo)
   do  a1[mayor] > nuevo →
       poner nuevo en el lugar correcto en a1, descartando el viejo a1[mayor]
       P2 ! (a1[mayor]);
       P2 ? (nuevo);
    od
}

Process P2
{  int nuevo, a2[1:n/2]; const menor = 1;
   ordenar a2 en orden no decreciente
   P1 ? (nuevo);
   P1 ! (a2[menor]);
   do  a2[menor] < nuevo →
       poner nuevo en el lugar correcto en a2, descartando el viejo a2[menor]
       P1 ? (nuevo);
       P1 ? (a2[menor]);
    od
}
```

Ejemplo

Ordenación de un Arreglo

- Notar que en la implementación del intercambio, las sentencias de entrada y salida son bloqueantes → usamos una solución asimétrica.

- Para evitar *deadlock*, *P1* primero ejecuta una salida y luego una entrada, y *P2* ejecuta primero una entrada y luego una salida.

- ***Comunicación guardada para programar una solución simétrica:*** esta solución es más costosa de implementar.

```
P1: … if P2 ? (nuevo) → P2 ! (a1[mayor])
      □ P2 ! (a1[mayor]) → P2 ? (nuevo)
    fi…

P2: … if P1 ? (nuevo) → P1 ! (a2[menor])
      □ P1 ! (a2[menor]) → P1 ? (nuevo)
    fi …
```

- Mejor caso → los procesos intercambian solo un par de valores.

- Peor caso → intercambian n/2 + 1 valores: n/2 para tener cada valor en el proceso correcto y uno para detectar terminación.

Ejemplo

Ordenación de un Arreglo

- Solución con ***b*** procesos ***P[1:b]***, inicialmente con ***n/b*** valores cada uno.
- Cada uno primero ordena sus ***n/b*** valores. Luego ordenamos los ***n*** elementos usando aplicaciones paralelas repetidas del algoritmo compare-and-exchange
- Cada proceso ejecuta una serie de rondas:
  - En las impares, cada proceso con número impar juega el rol de ***P1***, y cada proceso con número par el de ***P2***.
  - En las rondas pares, cada proceso numerado par juega el rol de ***P1***, y cada proceso impar el rol de ***P2***.
- **Ejemplo:** *b=4* - Algoritmo odd/even exchange sort

ronda	**P[1]**	**P[2]**	**P[3]**	**P[4]**
0	8	7	6	5
1	7	8	5	6
2	7	5	8	6
3	5	7	6	8
4	5	6	7	8

Ejemplo

Ordenación de un Arreglo

- Cada intercambio progresa hacia una lista totalmente ordenada.
- ¿Cómo pueden detectar los procesos si toda la lista está ordenada?. Un proceso individual no puede detectar que la lista entera está ordenada después de una ronda pues conoce solo dos porciones:
  - Se puede usar un coordinador separado. Después de cada ronda, los procesos le dicen a éste si hicieron algún cambio a su porción.
    - ✓ ***2b*** mensajes de overhead en cada ronda.
  - Que cada proceso ejecute suficientes rondas para garantizar que la lista estará ordenada (en general, al menos ***b*** rondas).
    - ✓ Cada proceso intercambia hasta ***n/b+1*** mensajes por ronda.
    - ✓ El algoritmo requiere hasta $b^2*(n/b + 1)$ intercambio de mensajes.

Paradigmas de Interacción entre Procesos

Paradigmas para la interacción entre procesos

- 3 esquemas básicos de interacción entre procesos: *productor/consumidor*, *cliente/servidor* e *interacción entre pares*.
- Estos esquemas básicos se pueden combinar de muchas maneras, dando lugar a otros **paradigmas** o modelos de interacción entre procesos.

Paradigma 1: *master / worker*
Implementación distribuida del modelo *Bag of Task*.

Paradigma 2: *algoritmos heartbeat*
Los procesos periódicamente deben intercambiar información con mecanismos tipo send/receive.

Paradigma 3: *algoritmos pipeline*
La información recorre una serie de procesos utilizando alguna forma de receive/send.

Paradigmas para la interacción entre procesos

Paradigma 4: *probes (send) y echoes(receive)*
La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información.

Paradigma 5: *algoritmos broadcast*
Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas.

Paradigma 6: *token passing*
En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. También permite la toma de decisiones distribuidas.

Paradigma 7: *servidores replicados*
Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos.

Paradigmas para la interacción entre procesos

Manager/Worker

- El concepto de ***bag of tasks*** usando variables compartidas supone que un conjunto de workers comparten una "bolsa" con tareas independientes. Los workers sacan una tarea de la bolsa, la ejecutan, y posiblemente crean nuevas tareas que ponen en la bolsa (ejemplo en LINDA manejando un espacio compartido de tuplas).

- La mayor virtud de este enfoque es la escalabilidad y la facilidad para equilibrar la carga de trabajo de los workers.

- Analizaremos la implementación de este paradigma con mensajes en lugar de MC. Para esto un proceso *manager* implementará la "bolsa" manejando las tasks, comunicándose con los workers y detectando fin de tareas. **Se trata de un esquema C/S**.

- Ejemplo: multiplicación de matrices ralas.

Paradigmas para la interacción entre procesos

Heartbeat

- Paradigma *heartbeat* ⇒ útil para soluciones iterativas que se quieren paralelizar.
- Usando un esquema "*divide & conquer*" se distribuye la carga (datos) entre los workers; cada uno es responsable de actualizar una parte.
- Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos.
- Cada "paso" debiera significar un progreso hacia la solución.
- Formato general de los worker:

```
process worker [i =1 to numWorkers]
  {  declaraciones e inicializaciones locales;
     while (no terminado)
       {  send valores a los workers vecinos;
          receive valores de los workers vecinos;
          Actualizar valores locales;
       }
  }
```

- Ejemplo: grid computations (imágenes), autómatas celulares (simulación de fenómenos como incendios o crecimiento biológico).

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

Los procesadores están conectados por canales bidireccionales. Cada uno se comunica sólo con sus vecinos y conoce esos links.

¿Cómo puede cada procesador determinar la topología completa de la red?

- Modelización:
  - Procesador ⇒ proceso
  - Links de comunicación ⇒ canales compartidos.
- Soluciones: los vecinos interactúan para intercambiar información local.

Algoritmo Heartbeat: se expande enviando información; luego se contrae incorporando nueva información.

- Procesos ***Nodo[p:1..n]***.
- Vecinos de ***p***: ***vecinos[1:n]*** → vecinos[q] es true si q es vecino de p.
- ***Problema:*** computar ***top*** (matriz de adyacencia), donde ***top[p,q]*** es true si p y q son vecinos.

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

Cada nodo debe ejecutar un nº de rondas para conocer la topología completa. Si el diámetro ***D*** de la red es conocido se resuelve con el siguiente algoritmo.

```
chan topologia[1:n] ([1:n,1:n] bool)

Process Nodo[p:1..n]
{  bool vecinos[1:n],  bool nuevatop[1:n,1:n],  top[1:n,1:n] = ([n*n] false);
   top[p,1..n] = vecinos;

   for (r =0 ; r < D; r++)
      { for [q = 1 to n st vecinos[q] ] send topologia[q](top);
         for [q = 1 to n st vecinos[q] ]
           {  receive topologia[p](nuevatop);
              top = top or nuevatop;
            }
      }
 }
```

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

- Rara vez se conoce el valor de ***D***.
- Excesivo intercambio de mensajes ⇒ los procesos cercanos al "centro" conocen la topología más pronto y no aprenden nada nuevo en los intercambios.
- El tema de la terminación ⇒ ¿local o distribuida?
- ***¿Cómo se pueden solucionar estos problemas?***
  - Después de ***r*** rondas, ***p*** conoce la topología a distancia ***r*** de él. Para cada nodo ***q*** dentro de la distancia ***r*** de ***p***, los vecinos de ***q*** estarán almacenados en la fila ***q*** de ***top*** ⇒ ***p*** ejecutó las rondas suficientes tan pronto como cada fila de ***top*** tiene algún valor ***true***.
  - Luego necesita ejecutar una última ronda para intercambiar la topología con sus vecinos.
- No siempre la terminación se puede determinar localmente.

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool)

Process Nodo[p:1..n]
{  bool vecinos[1:n], activo[1:n] = vecinos, top[1:n,1:n] = ([n*n]false), nuevatop[1:n,1:n];
   bool qlisto, listo = false;
   int emisor;
   top[p,1..n] = vecinos;
   while (not listo)
     {   for [q = 1 to n st activo[q] ] send topologia[q](p,false,top);
         for [q = 1 to n st activo[q] ]
            {   receive topologia[p](emisor,qlisto,nuevatop);
                top = top or nuevatop;
                if (qlisto) activo[emisor] = false;
             }
          if (todas las filas de top tiene 1 entry true) listo=true;
     }
    for [q = 1 to n st activo[q] ] send topologia[q](p,listo,top);
    for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop);
}
```

Paradigmas para la interacción entre procesos
Pipeline

- Un pipeline es un arreglo lineal de procesos "filtro" que reciben datos de un puerto (canal) de entrada y entregan resultados por un canal de salida.
- Estos procesos ("workers") pueden estar en procesadores que operan en paralelo, en un primer esquema *a lazo abierto* (W_1 en el INPUT, W_n en el OUTPUT).
- Un segundo esquema es el pipeline *circular*, donde W_n se conecta con W_1. Estos esquemas sirven en procesos iterativos o bien donde la aplicación no se resuelve en una pasada por el pipe.
- En un tercer esquema posible (*cerrado*), existe un proceso coordinador que maneja la "realimentación" entre W_n y W_1.
- Ejemplo: multiplicación de matrices en bloques.

Paradigmas para la interacción entre procesos

Probe-Echo

- Arboles y grafos son utilizados en muchas aplicaciones distribuidas como búsquedas en la WEB, BD, sistemas expertos y juegos.
- Las arquitecturas distribuidas se pueden asimilar a los nodos de grafos y árboles, con canales de comunicación que los vinculan.
- DFS es uno de los paradigmas secuenciales clásicos para visitar todos los nodos en un árbol o grafo. Este paradigma es el análogo concurrente de DFS.
- ***Prueba-eco*** se basa en el envío de un mensajes (“probe”) de un nodo al sucesor, y la espera posterior del mensaje de respuesta (“echo”).
- Los **probes** se envían en paralelo a todos los sucesores.
- Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (o no se conoce) un número fijo de nodos activos (ejemplo: redes móviles).

Paradigmas para la interacción entre procesos
Broadcast

- En la mayoría de las LAN cada procesador se conecta directamente con los otros. Estas redes normalmente soportan la primitiva ***broadcast***:

broadcast ch(m);

- Los mensajes broadcast de un proceso se encolan en los canales en el orden de envío, pero broadcast no es atómico y los mensajes enviados por procesos A y B podrían ser recibidos por otros en distinto orden.

- Se puede usar broadcast para diseminar información o para resolver problemas de sincronización distribuida. Ejemplo: semáforos distribuidos, la base es un ***ordenamiento total de eventos de comunicación*** mediante el uso de ***relojes lógicos***.

Paradigmas para la interacción entre procesos
Token Passing

- Un paradigma de interacción muy usado se basa en un tipo especial de mensaje (“token”) que puede usarse para otorgar un permiso (control) o recoger información global de la arquitectura distribuida. Un ejemplo del primer tipo de algoritmos es el caso de tener que controlar ***exclusión mutua distribuida***.
- Ejemplos de recolección de información de estado son los algoritmos de detección de terminación en computación distribuida.
- Aunque el problema de la SC se da principalmente en programas de MC, puede encontrarse en programas distribuidos cuando hay algún recurso compartido que puede usar un único proceso a la vez. Generalmente es una componente de un problema más grande, tal como asegurar consistencia en un sistema de BD.
- Soluciones posibles: Monitor activo que da permiso de acceso (ej: locks en archivos), semáforos distribuidos (usando broadcast, con gran intercambio de mensajes), o *token ring* (descentralizado y fair).

Paradigmas para la interacción entre procesos

Servidores Replicados

- Un server puede ser replicado cuando hay múltiples instancias de un recurso: cada server maneja una instancia.
- La replicación también puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.
- Ejemplo: problema de los filósofos
  - Modelo ***centralizado*****:** los Filósofo se comunican con ***UN*** proceso Mozo que decide el acceso o no a los recursos.
  - Modelo ***distribuido*****:** supone **5 procesos Mozo,** cada uno manejando un tenedor. Un Filósofo puede comunicarse con **2** Mozos (izquierdo y derecho), solicitando y devolviendo el recurso. Los Mozos **NO se comunican entre ellos**.
  - Modelo ***descentralizada*****:** cada Filósofo ve **un único** Mozo. Los Mozos se comunican entre ellos (cada uno con sus **2** vecinos) para decidir el manejo del recurso asociado a "su" Filósofo.